Gregor Miller
gregor{at}ece.ubc.ca

OpenGL Programming

Background
Conventions
Simple Program

SGI & GL

- Silicon Graphics revolutionized graphics in 1982 by implementing the pipeline in hardware
- System was accessed through API called *GL*
- Much simpler than before to implement interactive 3D applications

OpenGL

- Success of GL led to OpenGL (1992), a platform-independent API that was:
  - Easy to use
  - Level of abstraction high enough for simple implementation but low enough to be close to hardware to get better performance
  - Focus on rendering, no device contexts or any OS specific dependencies

OpenGL Review

- Originally controlled by Architecture Review Board (ARB)
  - Extensions named after this
  - ARB replaced by Kronos
- Current version 4.2, previous versions stable
- OpenGL ES 1.0 and 2.0 (Embedded Systems) in iPhone and PS3

OpenGL Libraries

- Core library
  - OpenGL32.lib on Windows
  - libGL.a on *nix systems
- Utility Library (GLU)
  - Provides higher-level functionality using the Core
  - Camera setup, etc.
- OS GUI Drivers
  - GLX, WGL, AGL (useful for extensions)

GLUT

- OpenGL Utility Toolkit (GLUT)
- Provides OS-level functionality in a cross-platform API
  - Open a window, set up graphics context
  - Get input from mouse and keyboard
  - Menus
  - Event-driven (common to GUIs)
- Code is portable but options are limited in scope

OpenGL Functions

- Primitives
  - Points
  - Lines
    - Triangles / Quads / Polygons
- Attributes
  - Colour / Texture / Transparency

OpenGL Functions

- Transformations
  - Viewing
  - Modelling
- Control (GLUT)
- Input (GLUT)
- Query (platform specific)

OpenGL State

- OpenGL is a state machine
- Functions are of two types
  - Primitive generating
    - Produces output if primitives are visible
    - Processing of vertices and appearance of primitives controlled by the state
  - State changing
    - Transformation and Attribute functions

C-Style Constructs

- No Object Oriented Programming techniques used
- Multiple functions in the API for one logical command
  - `glVertex3f`
  - `glVertex2i`
  - `glVertex3dv`
- Underlying storage mode is the same
- Can overload these in C++ - care needs to be taken to ensure efficiency is not reduced

OpenGL Function Format

Function

Dimensions

Format (floats)

```
glVertex3f(x,y,z)
```

Library Prefix

Data

OpenGL Function Format

Function

Dimensions

Format (floats)

```
gl Vertex 3 f v (p)
```

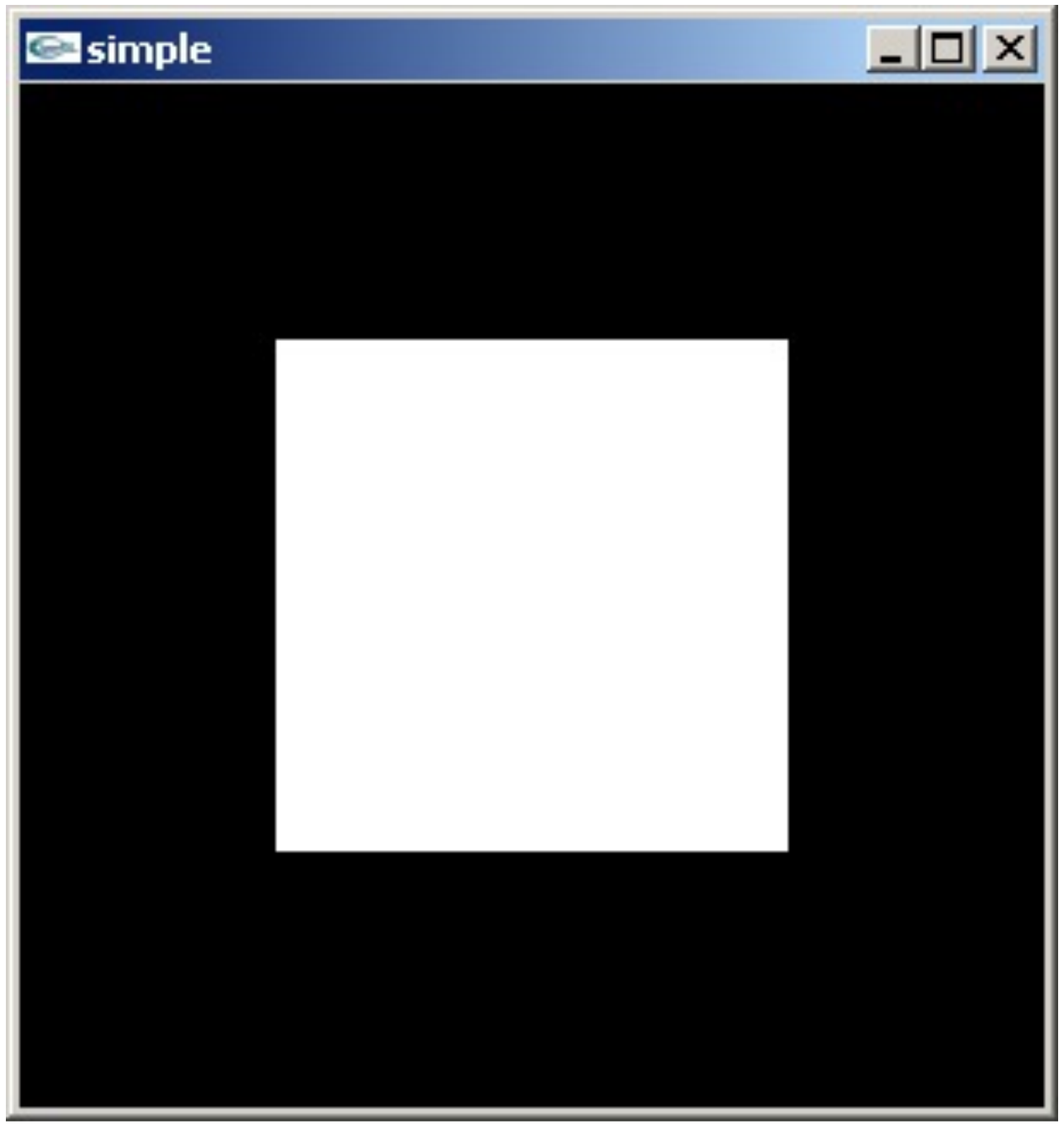
Library Prefix

Data

Array

OpenGL Headers

- The API is declared in
  - **`gl.h`, `glu.h`** & **`glut.h`** (usually inside a GL directory on the include path)
  - **`#include <GL/glut.h>`**
    - E.g.
      - **`glBegin(GL_TRIANGLES);`**
      - **`glClear(GL_COLOR_BUFFER_BIT);`**
- Headers also define types such as GLfloat, GLdouble (notice case)



A Simple Program

Generate a square on a solid background

simple.c

```
#include <GL/glut.h>
void mydisplay()
{
      glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0.5, 0.5);
                glVertex2f(0.5, 0.5);
                glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
}

int main(int argc, char** argv)
{
        glutCreateWindow("simple");
        glutDisplayFunc(mydisplay);
        glutMainLoop();
}
```

Event Loop

- Like many GUI systems, GLUT uses an *event-based* system for defining applications
- Callbacks are function pointers, usually associated with particular tasks
- The program defines a *display callback* function called **`mydisplay()`**
- Every GLUT program must have a display callback
- The callback is executed whenever the OS wishes to refresh the display e.g. when the window is opened or unoccluded
- **`main()`** ends with the program entering an infinite loop which processes events

Defaults

- **simple.c** uses many default values for OpenGL
  - Viewing
  - Colours
  - Window parameters
- Need to define more variables to exert greater control over system

Compilation

- Windows (Visual Studio)
  - Get GLUT from web (www.opengl.org has links)
    - glut.h, glut32.lib, glut32.dll
  - Create a console application
  - Add path to headers / libs in Visual Studio options
  - Add opengl32.lib, glu32.lib and glut32.lib to Link Input (under project settings)

Compilation

- Mac (Xcode)
  - Create Cocoa Application
  - Add OpenGL.Framework and GLUT.Framework
  - Remove Objective-C binding (clear GCC_PREFIX_HEADER attribute of Target settings)
  - http://blog.onesadcookie.com/2007/12/xcodeglut-tutorial.html

Compilation

- *nix
  - Headers usually in .../include/GL/
  - Link with -lglut -lglu -lgl flags
  - May have to include X libraries
  - Mesa included with Linux (can also install Nvidia/ATi Linux drivers)
  - Check web for more information

gluX

- Extension support system
- Cross platform extension management
- User-friendly (for developers and users)

- Might look into this in later lectures...

Glux

- C++ based
- Supports texture I/O
- Flexible windowing system
- Supports more options than GLUT (e.g. threading, user-control of event loop)
- Immature, under active development
- This course will continue to use GLUT, but feel free to try others

OpenGL Programming II

Complete graphics programs
Introduction to viewing
Fundamental OpenGL primitives

Program Structure

- GLUT programs have a very similar structure
  - **main()**:
    - Defines callbacks
      - **Display()**, **Reshape()**, inputs etc.
    - Initializes rendering context and opens window
    - Enters event loop (unless using freeGLUT or Glux)
- **Init()**: sets state variables for initial display
  - Attributes
  - Lighting
  - Buffers

Extending simple.c

- Refine simple.c so that the same output is produced, but defaults are set by us
- Defaults will be set for:
  - Colour
  - Viewing
  - Window properties

main(int argc, char **argv)

```
#include <GL/glut.h>

...

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(0, 0);
   glutCreateWindow("simple");
   glutDisplayFunc(&Display);

   Init();

   glutMainLoop();
}
```

GLUT Functions

- **glutInit()** gives the application command line arguments, and initializes the GLUT subsystem
- **glutInitDisplayMode()** requests properties for the rendering context
  - RGB colour
  - Single buffering
- **glutWindowSize()** in pixels
- **glutWindowPosition()** in pixels, from top-left of display
- **glutCreateWindow()** creates a window with the title “simple”
- **glutDisplayFunc()** sets the function to call for rendering (the display callback)
- **glutMainLoop()** enters the infinite loop for event handling
  - Good for simple programs, bad for anything moderately complex (like a game)

Init()

```
void Init()
{
   glClearColor (0.0, 0.0, 0.0, 1.0);

   glColor3f(1.0, 1.0, 1.0);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();

   glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

Black

Opaque

White

Matrix Stack

View Volume

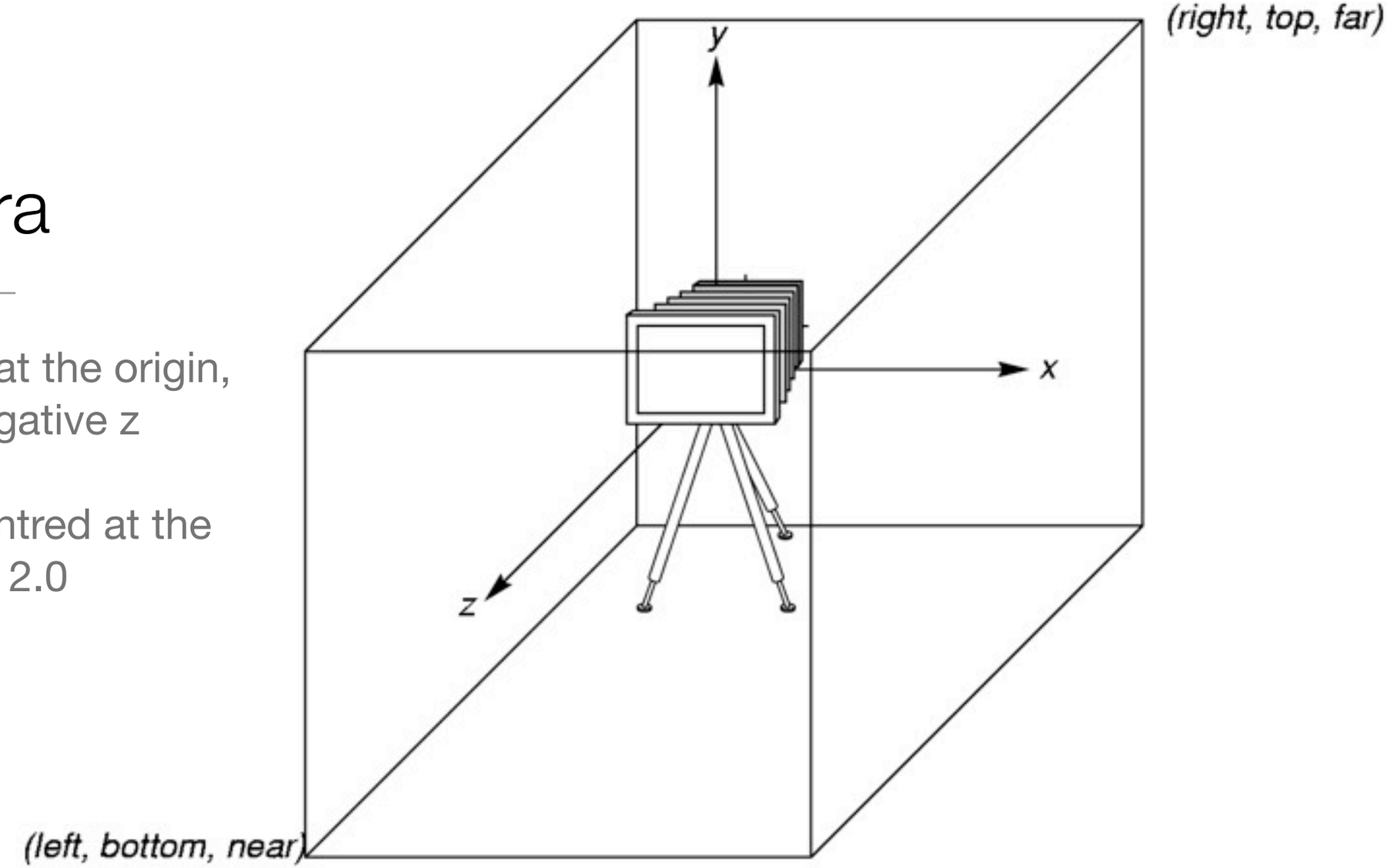
Coordinate Systems

- The units in **glVertex** are determined by the application and are called *object coordinates*
- The viewing specification is also in object coordinates.
- The size of the view volume determines what will appear in the image
- OpenGL converts to *camera* coordinates and then to *screen* coordinates
- OpenGL also uses internal representations usually not exposed through the API
  - For example, reading the depth buffer will not return depth values in object coordinates

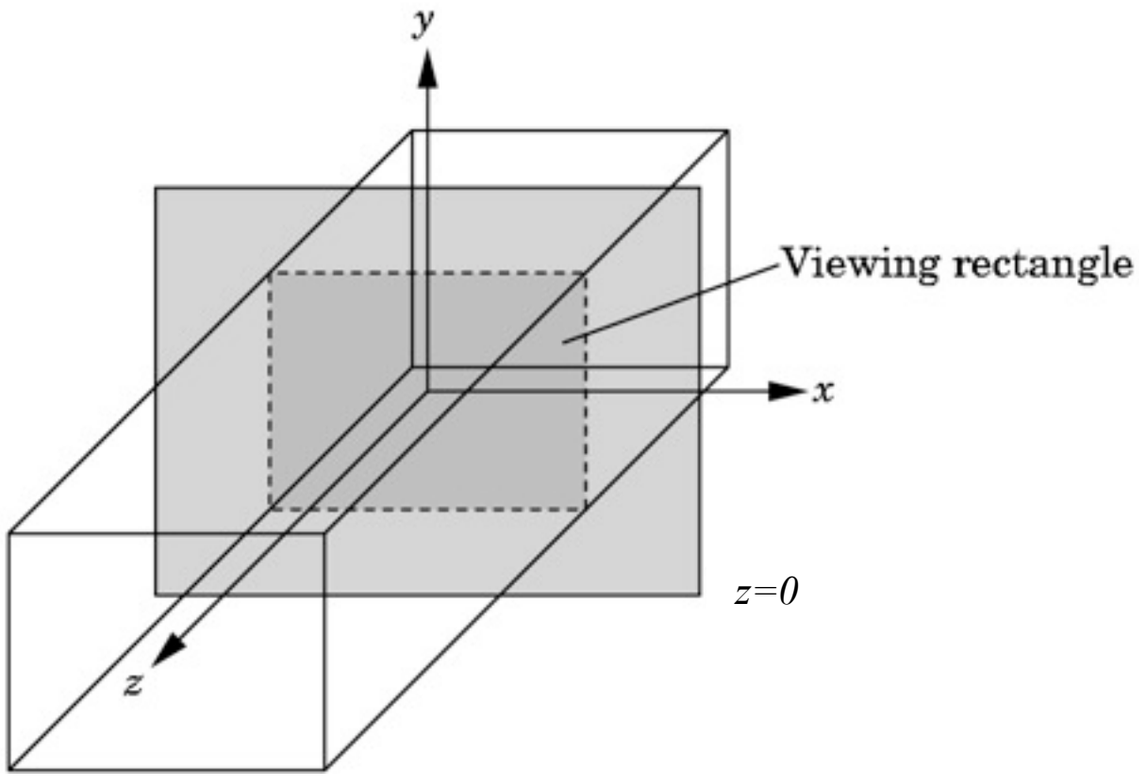
OpenGL Camera

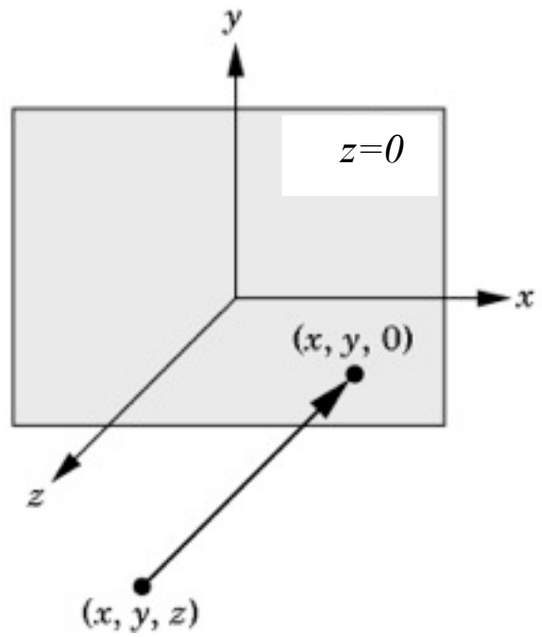
Default camera is placed at the origin, pointing in direction of negative z

Default view volume is centred at the origin with sides of length 2.0



Orthographic Viewing





In the default orthographic view, points are projected forward along the *z* axis onto the plane *z = 0*

Transformations and Viewing

- Projection in OpenGL is performed using the *projection* matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first:
  - **glMatrixMode(GL_PROJECTION);**
- Transformation functions combine via matrix multiplication, so we start with the identity:
  - **glLoadIdentity();**
- Then we supply the matrix which defines the view volume
  - **glOrtho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f);**

2D and 3D Viewing

- In **glOrtho(left, right, bottom, top, near, far)** the near and far distances are measured *from* the camera

- 2D vertex commands place all vertices in the plane *z=0*

- If the application is 2D, we can use the function

  - **gluOrtho2D(left, right, bottom, top)**

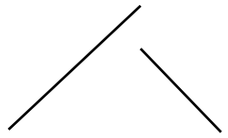
- In 2D the view volume becomes a *clipping window*

Display()

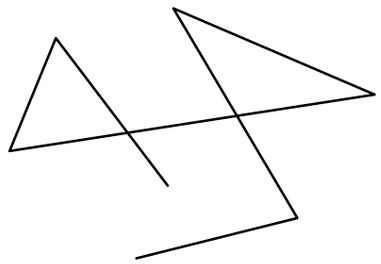
```
void Display()
{
 glClear(GL_COLOR_BUFFER_BIT);
 glBegin(GL_POLYGON);
                          2D Position
     glVertex2f(-0.5, -0.5);
     glVertex2f(-0.5, 0.5);
     glVertex2f(0.5, 0.5);
     glVertex2f(0.5, -0.5);
 glEnd();
 glFlush();
}
```

OpenGL Primitives

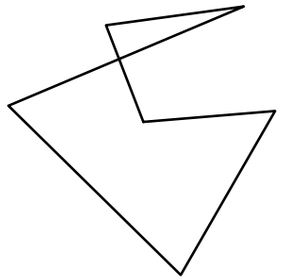
GL_POINTS



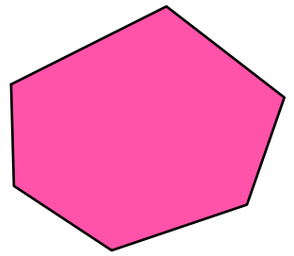
GL_LINES



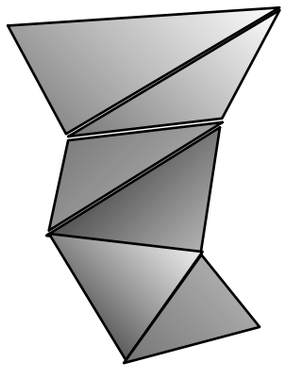
GL_LINE_STRIP



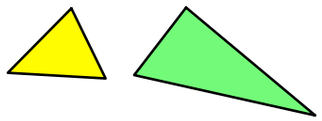
GL_LINE_LOOP



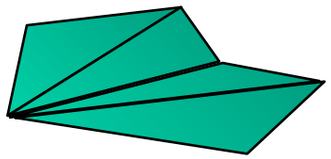
GL_POLYGON



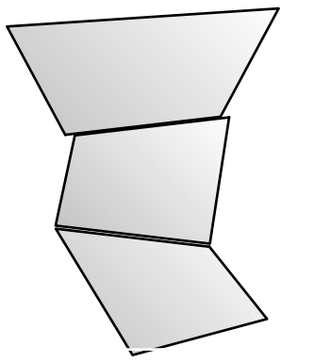
GL_TRIANGLE_STRIP



GL_TRIANGLES



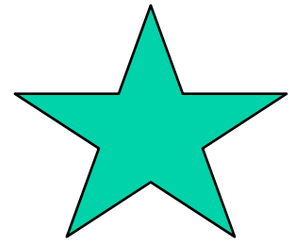
GL_TRIANGLE_FAN



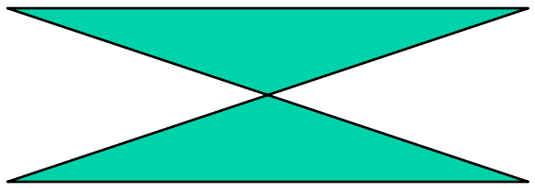
GL_QUAD_STRIP

Polygon Issues

- OpenGL will only display polygons correctly if they are:
  - *Simple*: edges cannot cross
  - *Convex*: All points on a line connecting two points in the polygon are also in the polygon
  - *Flat*: all vertices are in the same plane
- Triangles satisfy all of these conditions



Non-convex polygon



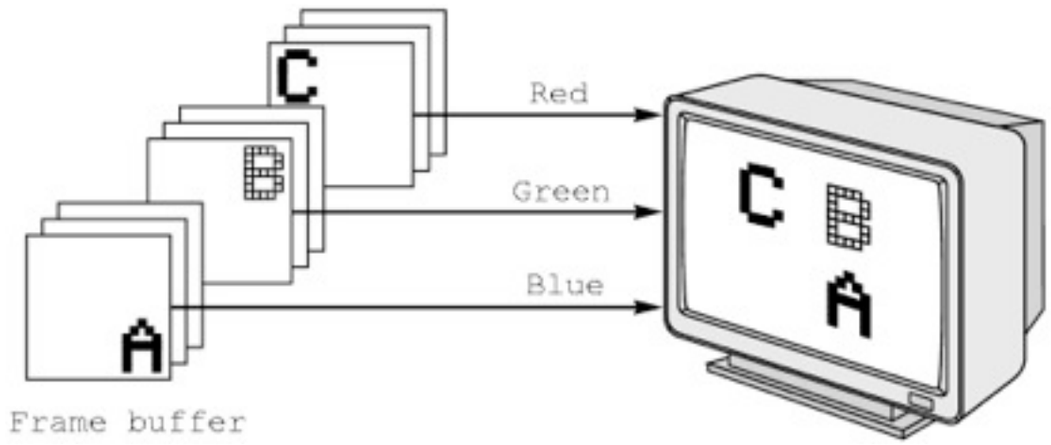
Non-simple polygon

Attributes

- Attributes are part of the OpenGL state and determine the appearance of objects
  - Colour (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
- Polygon mode
  - Display as filled: solid colour or stipple pattern
  - Display edges
  - Display vertices

RGB Colour

- Each colour component is stored separately in the frame buffer
- Usually 8 bits per component
- In **glColor3f()** the values range from 0.0 (none) to 1.0 (all)
- In **glColor3ub()** the values range from 0 (none) to 255 (all) (8-bits)



Colour and State

- When the colour is set by **glColor** it becomes part of the state and everything will use that colour until the state is changed

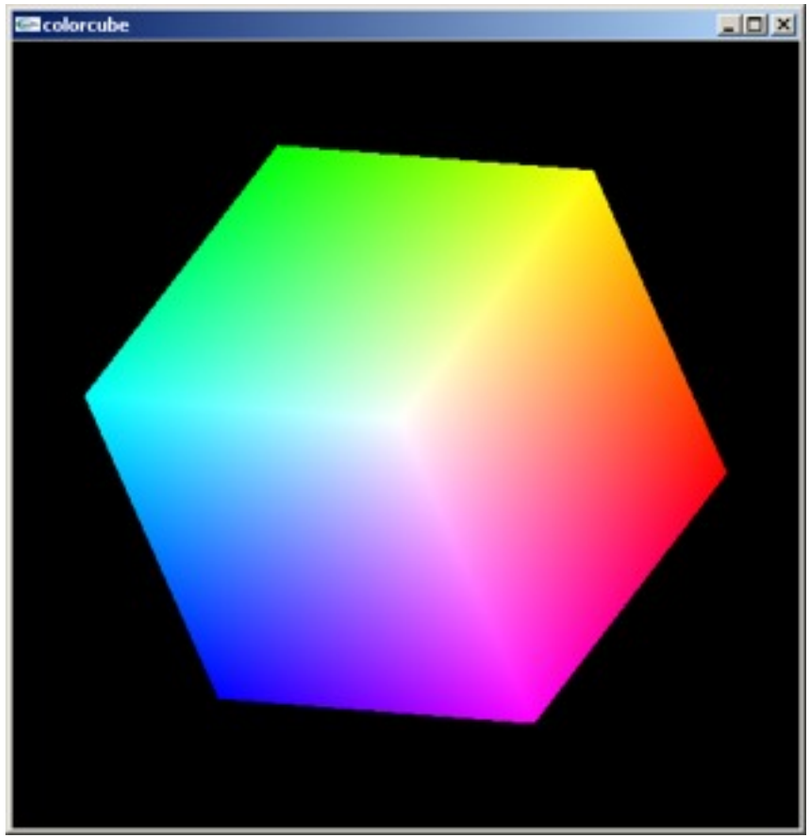
  - Colours and other attributes are not part of the object but are assigned when the the object is rendered

- We can create conceptual vertex colours using code such as:

```
glColor3f(...);
glVertex3f(...);
glColor3f(...);
glVertex3f(...);
```

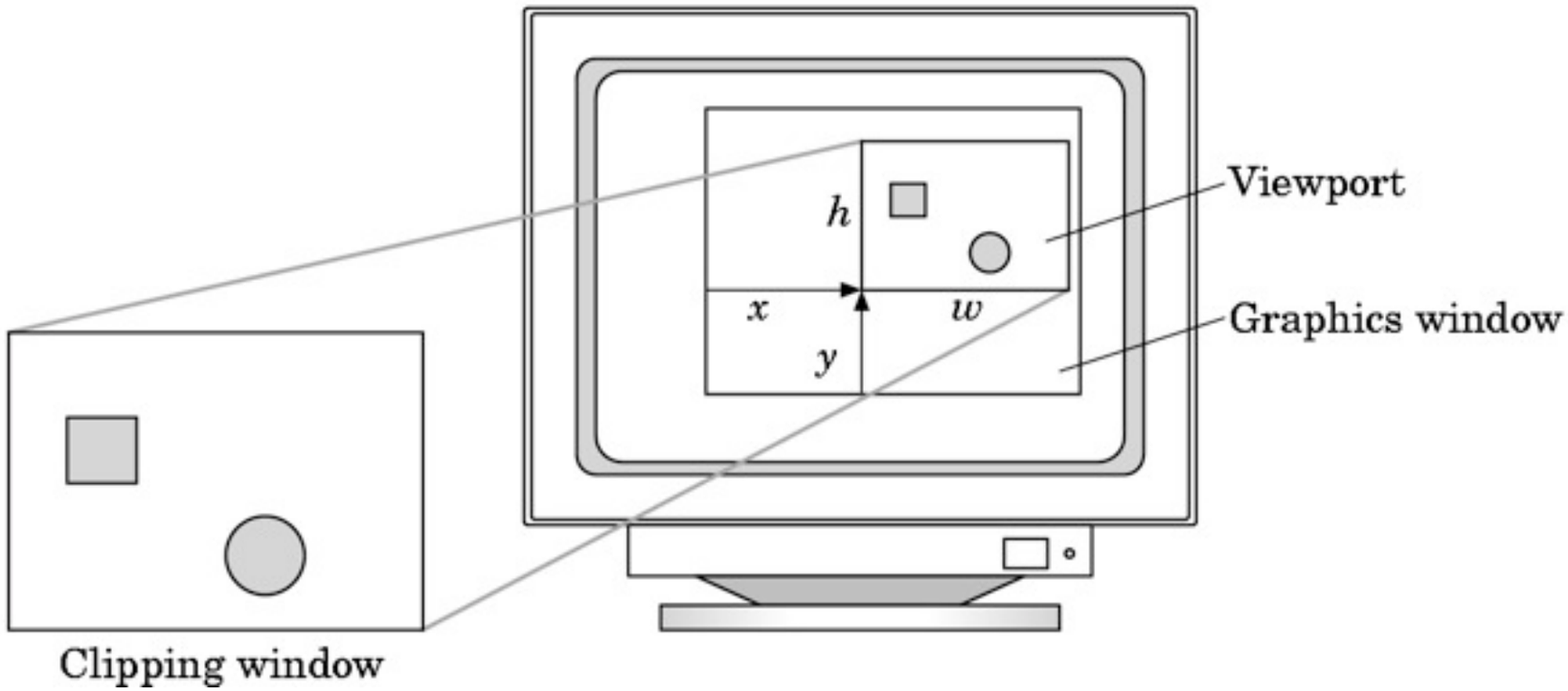
Smooth Colour

- Default is *smooth* shading
  - OpenGL interpolates vertex colours across visible polygons
- Alternative is flat shading
  - Colour of first vertex determines fill colour
- **glShadeModel(GL_SMOOTH)** or
- **glShadeModel(GL_FLAT)**



Viewports

- Don’t have to use the entire window for the image:
  - **glViewport(x, y, w, h)**
- Values in pixels (corresponding to screen coordinates)



Input and interaction

Input devices
Event based interfaces
Double buffering
Event input with GLUT

Input Modes

- Input devices contain a trigger which sends a signal to the OS
  - Mouse button
  - Key press or release
- When triggered, input devices send information to the system
  - Position and state of mouse cursor
  - ASCII code of key
- Input provided to program only when user triggers device

Event model

- Most systems have more than one input device with arbitrary timing
- Each trigger generates an *event* whose information is put in the *event queue* for use by the user's program
- Various event types:
  - Window - resize, move, minimize
  - Mouse - click, move
  - Keyboard - press, release
  - Idle - define what to do when no events are queued

Callbacks

- Programming interface for event-driven input
- Define a *callback function* for each type of event
- This user-supplied function is executed when the event occurs

- GLUT example:
  - **glutMouseFunc(&Mouse);**
  - **Mouse()** is our function for interpreting mouse events

GLUT Callbacks

- GLUT recognizes a subset of OS GUI events
  - **glutDisplayFunc()**
  - **glutMouseFunc()**
  - **glutReshapeFunc()**
  - **glutKeyboardFunc()**
  - **glutIdleFunc()**
  - **glutMotionFunc()**
  - **glutPassiveMotionFunc()**

GLUT Event Loop

- Recall the last line in **main(...)** for a program using GLUT is
  - **glutMainLoop()**
- This puts the program into an infinite loop for event processing

- In each pass through the event loop, GLUT:
  - Looks at events in the queue
  - For each event, GLUT executes the corresponding callback function
  - If no callback is defined, the event is ignored

The Display Callback

- The display callback is executed whenever GLUT determines the window should be refreshed:
  - When the window is first opened
  - When the window is reshaped
  - When a window is exposed
  - When the user program instructs GLUT to refresh the display
- In **main(...)**
  - **glutDisplayFunc(&Display())** identifies the function to be executed
  - Every GLUT program must have a display callback

Instructing the OS to Refresh

- Many events can invoke the display callback
    - This can lead to multiple executions of the display callback on a single frame render
- Avoid this problem using **glutPostRedisplay()**
- This sets a flag which GLUT then checks for
- If the flag is set, the display callback is executed by GLUT

Animating a Display

- When we redraw the display with the callback, we usually start by clearing the window with **glClear()**
- Then we draw the new frame
- Unfortunately the framebuffer is simultaneously read from by the display and written to by OpenGL, leading to flickering
- Instead of using a single buffer, we use two (double buffering):
  - Front Buffer: one that is displayed but not written to
  - Back Buffer: one that is written to but not displayed
- Request double buffering in **main(...)** with:
  - **glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE)**

Double Buffering

At the end of the display callback the buffers are swapped

```
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT|….)
.
/* draw graphics here */
.
        glutSwapBuffers();
}
```

Using the Idle callback

- The idle callback is executed whenever there are no events in the event queue
  - **glutIdleFunc(&Idle)**
  - Useful for animations:

```
void Idle() {
/* change something */
        t += dt
        glutPostRedisplay();
}

Void Display() {
        glClear();
/* draw something that depends on t */
        glutSwapBuffers();
}
```

Using Globals

- The form of all GLUT callbacks is fixed:
  - **void Display()**
  - **void Mouse(GLint button, GLint state, GLint x, GLint y)**
- Must use globals to pass information to callbacks
- Can work around this using C++, and wrapper methods in C
  - Can use alternatives such as freeGLUT and Glux if OOP is important for your application